Pig Latin: A Not-So-Foreign Language for Data Processing

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins
Yahoo! Research
{olston, breed, utkarsh, ravikumar, atomkins}@yahoo-inc.com

ABSTRACT

There is a growing need for ad-hoc analysis of extremely large data sets, especially at internet companies where innovation critically depends on being able to analyze terabytes of data collected every day. Parallel database products, e.g., Teradata, offer a solution, but are usually prohibitively expensive at this scale. Besides, many of the people who analyze this data are entrenched procedural programmers, who find the declarative, SQL style to be unnatural. The success of the more procedural *map-reduce* programming model, and its associated scalable implementations on commodity hardware, is evidence of the above. However, the map-reduce paradigm is too low-level and rigid, and leads to a great deal of custom user code that is hard to maintain, and reuse.

We describe a new language called *Pig Latin* that we have designed to fit in a sweet spot between the declarative style of SQL, and the low-level, procedural style of map-reduce. The accompanying system, Pig, is fully implemented, and compiles Pig Latin into physical plans that are executed over *Hadoop*, an open-source, map-reduce implementation. We give a few examples of how engineers at Yahoo! are using Pig to dramatically reduce the time required for the development and execution of their data analysis tasks, compared to using Hadoop directly. We also report on a novel debugging environment that comes integrated with Pig, that can lead to even higher productivity gains. Pig is an open-source, Apache-incubator project, and available for general use.

1. INTRODUCTION

At a growing number of organizations, innovation revolves around the collection and analysis of enormous data sets such as web crawls, search logs, and click streams. Internet companies such as Amazon, Google, Microsoft, and Yahoo! are prime examples. Analysis of this data constitutes the innermost loop of the product improvement cycle. For example, the engineers who develop search engine ranking algorithms spend much of their time analyzing search logs looking for exploitable trends.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

The sheer size of these data sets dictates that it be stored and processed on highly parallel systems, such as shared-nothing clusters. Parallel database products, e.g., Teradata, Oracle RAC, Netezza, offer a solution by providing a simple SQL query interface and hiding the complexity of the physical cluster. These products however, can be prohibitively expensive at web scale. Besides, they wrench programmers away from their preferred method of analyzing data, namely writing imperative scripts or code, toward writing declarative queries in SQL, which they often find unnatural, and overly restrictive.

As evidence of the above, programmers have been flocking to the more procedural *map-reduce* [4] programming model. A map-reduce program essentially performs a group-by-aggregation in parallel over a cluster of machines. The programmer provides a map function that dictates how the grouping is performed, and a reduce function that performs the aggregation. What is appealing to programmers about this model is that there are only two high-level declarative primitives (map and reduce) to enable parallel processing, but the rest of the code, i.e., the map and reduce functions, can be written in any programming language of choice, and without worrying about parallelism.

Unfortunately, the map-reduce model has its own set of limitations. Its one-input, two-stage data flow is extremely rigid. To perform tasks having a different data flow, e.g., joins, inelegant workarounds have to be devised. Also, custom code has to be written for even the most common operations, e.g., projection and filtering. These factors lead to code that is difficult to reuse and maintain, and in which the semantics of the analysis task are obscured. Moreover, the opaque nature of the map and reduce functions impedes the ability of the system to perform optimizations.

We have developed a new language called *Pig Latin* that combines the best of both worlds: high-level declarative querying in the spirit of SQL, and low-level, procedural programming à la map-reduce.

EXAMPLE 1. *Suppose we have a table* `urls: (url, category, pagerank)`*. The following is a simple SQL query that finds, for each sufficiently large category, the average pagerank of high-pagerank urls in that category.*

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10^6
```

An equivalent Pig Latin program is the following. (Pig Latin is described in detail in Section 3; a detailed understanding of the language is not required to follow this example.)

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)>10^6;
output = FOREACH big_groups GENERATE
            category, AVG(good_urls.pagerank);
```

As evident from the above example, a Pig Latin program is a sequence of steps, much like in a programming language, each of which carries out a single data transformation. This characteristic is immediately appealing to many programmers. At the same time, the transformations carried out in each step are fairly high-level, e.g., filtering, grouping, and aggregation, much like in SQL. The use of such high-level primitives renders low-level manipulations (as required in map-reduce) unnecessary.

In effect, writing a Pig Latin program is similar to specifying a query execution plan, thereby making it easier for programmers to understand and control how their data processing task is executed. To experienced system programmers, this method is much more appealing than encoding their task as an SQL query, and then coercing the system to choose the desired plan through optimizer hints. (Automatic query optimization has its limits, especially with uncataloged data, prevalent user-defined functions, and parallel execution, which are all features of our target environment; see Section 2.) Even with a Pig Latin optimizer in the future, users will still be able to request conformation to the execution plan implied by their program.

Pig Latin has several other unconventional features that are important for our setting of casual ad-hoc data analysis by programmers. These features include support for a flexible, fully nested data model, extensive support for user-defined functions, and the ability to operate over plain input files without any schema information. Pig Latin also comes with a novel debugging environment that is especially useful when dealing with enormous data sets. We elaborate on these features in Section 2.

Pig Latin is fully implemented by our system, *Pig*, and is being used by programmers at Yahoo! for data analysis. Pig Latin programs are currently compiled into map-reduce jobs that are executed using *Hadoop*, an open-source, scalable implementation of map-reduce. Alternative backends can also be plugged in. Pig is an open-source project in the Apache incubator[1], and is available for general use.

The rest of the paper is organized as follows. In the next section, we describe the various features of Pig, and the underlying motivation for each. In Section 3, we dive into the Pig Latin data model and language. In Section 4, we describe the current implementation of Pig. We describe our novel debugging environment in Section 5, and outline a few real usage scenarios of Pig in Section 6. Finally, we discuss related work in Section 7, and conclude.

[1] `http://incubator.apache.org/pig`

2. FEATURES AND MOTIVATION

The overarching design goal of Pig is to be appealing to experienced programmers for performing ad-hoc analysis of extremely large data sets. Consequently, Pig Latin has a number of features that might seem surprising when viewed from a traditional database and SQL perspective. In this section, we describe the features of Pig, and the rationale behind them.

2.1 Algebraic Language

As seen in Example 1, in Pig Latin, a user specifies a sequence of steps where each step specifies only a *single*, high-level data transformation. This is stylistically different from the SQL approach where the user specifies a set of declarative constraints that collectively define the result. While the SQL approach is good for non-programmers and/or small data sets, experienced programmers who must manipulate large data sets often prefer the Pig Latin approach. As one of our users says,

> "I much prefer writing in Pig [Latin] versus SQL. The step-by-step method of creating a program in Pig [Latin] is much cleaner and simpler to use than the single block method of SQL. It is easier to keep track of what your variables are, and where you are in the process of analyzing your data." – Jasmine Novak, Engineer, Yahoo!

Note that although Pig Latin programs are written in a step-by-step fashion, it is not necessary that the operations be executed in that same order. The use of high-level, relational-algebra-style primitives, e.g., `GROUP, FILTER`, still allows traditional database optimizations to be carried out, in cases where the system and/or user have enough confidence that query optimization can succeed.

For example, if one is interested in the set of urls of pages that are classified as spam, but have a high pagerank score. In Pig Latin, one can write:

```
spam_urls = FILTER urls BY isSpam(url);
culprit_urls = FILTER spam_urls BY pagerank > 0.8;
```

where `culprit_urls` contains the final set of urls that the user is interested in.

The above Pig Latin fragment suggests that we first find the spam urls through the function `isSpam`, and then filter them by `pagerank`. However, this might not be the most efficient method. In particular, `isSpam` might be an expensive user-defined function that analyzes the url's content for spaminess. Then, it will be much more efficient to filter the urls by pagerank first, and invoke `isSpam` only on the pages that have high pagerank.

With Pig Latin, this optimization opportunity is available to the system. On the other hand, if these filters were buried within an opaque map or reduce function, such reordering and optimization would effectively be impossible.

2.2 Quick Start and Interoperability

Pig is designed to support ad-hoc data analysis. If a user has a data file obtained, say, from a dump of the search engine logs, she can run Pig Latin queries over it directly.

She need only provide a function that gives Pig the ability to parse the content of the file into tuples. There is no need to go through a time-consuming data import process prior to running queries, as in conventional database management systems. Similarly, the output of a Pig program can be formatted in the manner of the user's choosing, according to a user-provided function that converts tuples into a byte sequence. Hence it is easy to use the output of a Pig analysis session in a subsequent application, e.g., a visualization or spreadsheet application such as Excel.

It is important to keep in mind that Pig is but one of many applications in the rich "data ecosystem" of a company like Yahoo! By operating over data residing in external files, and not taking over control over the data, Pig readily interoperates with other applications in the ecosystem.

The reasons that conventional database systems do require importing data into system-managed tables are three-fold: (1) to enable transactional consistency guarantees, (2) to enable efficient point lookups (via physical tuple identifiers), and (3) to curate the data on behalf of the user, and record the schema so that other users can make sense of the data. Pig only supports read-only data analysis workloads, and those workloads tend to be scan-centric, so transactional consistency and index-based lookups are not required. Also, in our environment users often analyze a temporary data set for a day or two, and then discard it, so data curating and schema management can be overkill.

In Pig, stored schemas are strictly optional. Users may supply schema information on the fly, or perhaps not at all. Thus, in Example 1, if the user knows the the third field of the file that stores the `urls` table is `pagerank` but does not want to provide the schema, the first line of the Pig Latin program can be written as:

```
good_urls = FILTER urls BY $2 > 0.2;
```

where $2 uses positional notation to refer to the third field.

2.3 Nested Data Model

Programmers often think in terms of nested data structures. For example, to capture information about the positional occurrences of terms in a collection of documents, a programmer would not think twice about creating a structure of the form `Map<documentId, Set<positions>>` for each `term`.

Databases, on the other hand, allow only flat tables, i.e., only atomic fields as columns, unless one is willing to violate the First Normal Form (1NF) [7]. To capture the same information about terms above, while conforming to 1NF, one would need to *normalize* the data by creating two tables:

```
term_info:  (termId, termString, ...)
position_info:  (termId, documentId, position)
```

The same positional occurence information can then be reconstructed by joining these two tables on `termId` and grouping on `termId, documentId`.

Pig Latin has a flexible, fully nested data model (described in Section 3.1), and allows complex, non-atomic data types such as set, map, and tuple to occur as fields of a table. There are several reasons why a nested model is more appropriate for our setting than 1NF:

- A nested data model is closer to how programmers think, and consequently much more natural to them than normalization.
- Data is often stored on disk in an inherently nested fashion. For example, a web crawler might output for each url, the set of outlinks from that url. Since Pig operates directly on files (Section 2.2), separating the data out into normalized form, and later recombining through joins can be prohibitively expensive for web-scale data.
- A nested data model also allows us to fulfill our goal of having an algebraic language (Section 2.1), where each step carries out only a single data transformation. For example, each tuple output by our `GROUP` primitive has one non-atomic field: a nested set of tuples from the input that belong to that group. The `GROUP` construct is explained in detail in Section 3.5.
- A nested data model allows programmers to easily write a rich set of user-defined functions, as shown in the next section.

2.4 UDFs as First-Class Citizens

A significant part of the analysis of search logs, crawl data, click streams, etc., is custom processing. For example, a user may be interested in performing natural language stemming of a search term, or figuring out whether a particular web page is spam, and countless other tasks.

To accommodate specialized data processing tasks, Pig Latin has extensive support for *user-defined functions* (UDFs). Essentially all aspects of processing in Pig Latin including grouping, filtering, joining, and per-tuple processing can be customized through the use of UDFs.

The input and output of UDFs in Pig Latin follow our flexible, fully nested data model. Consequently, a UDF to be used in Pig Latin can take non-atomic parameters as input, and also output non-atomic values. This flexibility is often very useful as shown by the following example.

EXAMPLE 2. *Continuing with the setting of Example 1, suppose we want to find for each category, the top 10 urls according to pagerank. In Pig Latin, one can simply write:*

```
groups = GROUP urls BY category;
output = FOREACH groups GENERATE
              category, top10(urls);
```

where top10() is a UDF that accepts a set of urls (for each group at a time), and outputs a set containing the top 10 urls by pagerank for that group.[2] *Note that our final output in this case contains non-atomic fields: there is a tuple for each category, and one of the fields of the tuple is the set of the top 10 urls in that category.*

Due to our flexible data model, the return type of a UDF does not restrict the context in which it can be used. Pig

[2] In practice, a user would probably write a more generic function than `top10()`: one that takes k as a parameter to find the top k tuples, and also the field according to which the top k must be found (`pagerank` in this example).

Latin has only one type of UDF that can be used in all the constructs such as filtering, grouping, and per-tuple processing. This is in contrast to SQL, where only scalar functions may be used in the `SELECT` clause, set-valued functions can only appear in the `FROM` clause, and aggregation functions can only be applied in conjunction with a `GROUP BY` or a `PARTITION BY`.

Currently, Pig UDFs are written in Java. We are building support for interfacing with UDFs written in arbitrary languages, including C/C++, Java, Perl and Python, so that users can stick with their language of choice.

2.5 Parallelism Required

Since Pig Latin is geared toward processing web-scale data, it does not make sense to consider non-parallel evaluation. Consequently, we have only included in Pig Latin a small set of carefully chosen primitives that can be easily parallelized. Language primitives that do not lend themselves to efficient parallel evaluation (e.g., the analogue of correlated subqueries in SQL) have been deliberately excluded.

Such operations can of course, still be carried out by writing UDFs. However, since the language does not provide explicit primitives for such operations, users are aware of how efficient their programs will be and whether they will be parallelized.

2.6 Debugging Environment

In any language, getting a data processing program right usually takes many iterations, with the first few iterations usually having some user-introduced erroneous processing. At the scale of data that Pig is meant to process, a single iteration can take many minutes or hours (even with large-scale parallel processing). Thus, the usual run-debug-run cycle can be very slow and inefficient.

Pig comes with a novel interactive debugging environment that generates a concise example data table illustrating the output of each step of the user's program. The example data is carefully chosen to resemble the real data as far as possible and also to fully illustrate the semantics of the program. Moreover, the example data is automatically adjusted as the program evolves.

This step-by-step example data can help in detecting errors early (even before the first iteration of running the program on the full data), and also in pinpointing the step that has errors. The details of our debugging environment are provided in Section 5.

3. PIG LATIN

In this section, we describe the details of the Pig Latin language. We describe our data model in Section 3.1, and the Pig Latin statements in the subsequent subsections. The emphasis of this section is not on the syntactical details of Pig Latin, but on how it meets the design goals and features laid out in Section 2. Also, this section only focusses on the language primitives, and not on how they can be implemented to execute in parallel over a cluster. Implementation is covered in Section 4.

$$\texttt{t} = \left(\texttt{'alice'}, \left\{ \begin{array}{c} (\texttt{'lakers', 1}) \\ (\texttt{'iPod', 2}) \end{array} \right\}, \left[\texttt{'age'} \rightarrow 20\right]\right)$$

Let fields of tuple `t` be called `f1, f2, f3`

Expression Type	Example	Value for `t`
Constant	`'bob'`	Independent of `t`
Field by position	`$0`	`'alice'`
Field by name	`f3`	['age' → 20]
Projection	`f2.$0`	{ ('lakers') ('iPod') }
Map Lookup	`f3#'age'`	20
Function Evaluation	`SUM(f2.$1)`	`1 + 2 = 3`
Conditional Expression	`f3#'age'>18? 'adult':'minor'`	`'adult'`
Flattening	`FLATTEN(f2)`	`'lakers', 1` `'iPod', 2`

Table 1: Expressions in Pig Latin.

3.1 Data Model

Pig has a rich, yet simple data model consisting of the following four types:

- *Atom*: An atom contains a simple atomic value such as a string or a number, e.g., `'alice'`.
- *Tuple*: A tuple is a sequence of *fields*, each of which can be any of the data types, e.g., `('alice', 'lakers')`.
- *Bag*: A bag is a collection of tuples with possible duplicates. The schema of the constituent tuples is flexible, i.e., not all tuples in a bag need to have the same number and type of fields, e.g.,

$$\left\{ \begin{array}{c} (\texttt{'alice', 'lakers'}) \\ \left(\texttt{'alice', ('iPod', 'apple')}\right) \end{array} \right\}$$

  The above example also demonstrates that tuples can be nested, e.g., the second tuple of the bag has a nested tuple as its second field.

- *Map*: A map is a collection of data items, where each item has an associated key through which it can be looked up. As with bags, the schema of the constituent data items is flexible, i.e., all the data items in the map need not be of the same type. However, the keys are required to be data atoms, mainly for efficiency of lookups. The following is an example of a map:

$$\left[\begin{array}{c} \texttt{'fan of'} \rightarrow \left\{ \begin{array}{c} (\texttt{'lakers'}) \\ (\texttt{'iPod'}) \end{array} \right\} \\ \texttt{'age'} \rightarrow 20 \end{array} \right]$$

  In the above map, the key `'fan of'` is mapped to a bag containing two tuples, and the key `'age'` is mapped to an atom 20.

  A map is especially useful to model data sets where schemas might change over time. For example, if web servers decide to include a new field while dumping logs, that new field can simply be included as an additional key in a map, thus requiring no change to existing programs, and also allowing access of the new field to new programs.

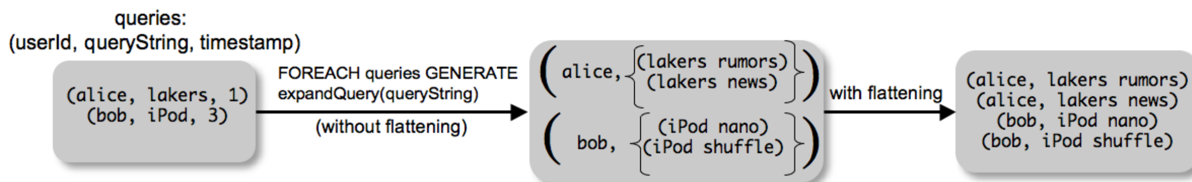

Figure 1: Example of *flattening* in FOREACH.

Table 1 shows the expression types in Pig Latin, and how they operate. (The *flattening* expression is explained in detail in Section 3.3.) It should be evident that our data model is very flexible and permits arbitrary nesting. This flexibility allows us to achieve the aims outlined in Section 2.3, where we motivated our use of a nested data model. Next, we describe the Pig Latin commands.

3.2 Specifying Input Data: LOAD

The first step in a Pig Latin program is to specify what the input data files are, and how the file contents are to be deserialized, i.e., converted into Pig's data model. An input file is assumed to contain a sequence of tuples, i.e., a bag. This step is carried out by the `LOAD` command. For example,

```
queries   =   LOAD 'query_log.txt'
              USING myLoad()
              AS (userId, queryString, timestamp);
```

The above command specifies the following:

- The input file is `query_log.txt`.
- The input file should be converted into tuples by using the custom `myLoad` deserializer.
- The loaded tuples have three fields named `userId`, `queryString`, and `timestamp`.

Both the `USING` clause (the custom deserializer) and the `AS` clause (the schema information) are optional. If no deserializer is specified, a default one, that expects a plain text, tab-delimited file, is used. If no schema is specified, fields must be referred to by position instead of by name (e.g., $0 for the first field). The ability to operate over plain text files, and the ability to specify schema information on the fly or not at all, allows the user to get started quickly (Section 2.2). To aid readability, it is desirable to include schemas while writing large Pig Latin programs.

The return value of a `LOAD` command is a *handle* to a bag which, in the above example, is assigned to the variable `queries`. This variable can then be used as an input in subsequent Pig Latin commands. Note that the `LOAD` command does not imply database-style loading into tables. Bag handles in Pig Latin are only *logical*—the `LOAD` command merely specifies what the input file is, and how it should be read. No data is actually read, and no processing carried out, until the user explicitly asks for output (see `STORE` command in Section 3.8).

3.3 Per-tuple Processing: FOREACH

Once input data file(s) have been specified through `LOAD`, one can specify the processing that needs to be carried out on the data. One of the basic operations is that of applying some processing to every tuple of a data set. This is achieved through the `FOREACH` command. For example,

```
expanded_queries = FOREACH queries GENERATE
                   userId, expandQuery(queryString);
```

The above command specifies that each tuple of the bag `queries` (loaded in the previous section) should be processed independently to produce an output tuple. The first field of the output tuple is the `userId` field of the input tuple, and the second field of the output tuple is the result of applying the UDF `expandQuery` to the `queryString` field of the input tuple. Suppose the UDF `expandQuery` generates a bag of likely expansions of a given query string. Then an example transformation carried out by the above statement is as shown in the first step of Figure 1.

Note that the semantics of the `FOREACH` command are such that there can be no dependence between the processing of different tuples of the input, thereby permitting an efficient parallel implementation. These semantics conform to our goal of only having parallelizable operations (Section 2.5).

In general, the `GENERATE` clause can be followed by a list of expressions that can be in any of the forms listed in Table 1. Most of the expression forms shown are straightforward, and have been included here only for completeness. The last expression type, i.e., *flattening*, deserves some attention.

Often, we want to eliminate nesting in data. For example, in Figure 1, we might want to flatten the set of possible expansions of a query string into separate tuples, so that they can be processed independently. We could also want to flatten the final result just prior to storing it. Nesting can be eliminated by the use of the `FLATTEN` keyword in the `GENERATE` clause. Flattening operates on bags by extracting the fields of the tuples in the bag, and making them fields of the tuple being output by `GENERATE`, thus removing one level of nesting. For example, the output of the following command is shown as the second step in Figure 1.

```
expanded_queries = FOREACH queries GENERATE
         userId, FLATTEN(expandQuery(queryString));
```

3.4 Discarding Unwanted Data: FILTER

Another very common operation is to retain only some subset of the data that is of interest, while discarding the rest. This operation is done by the `FILTER` command. For example, to get rid of bot traffic in the bag `queries`:

```
real_queries = FILTER queries BY userId neq 'bot';
```

The operator `neq` in the above example is used to signify

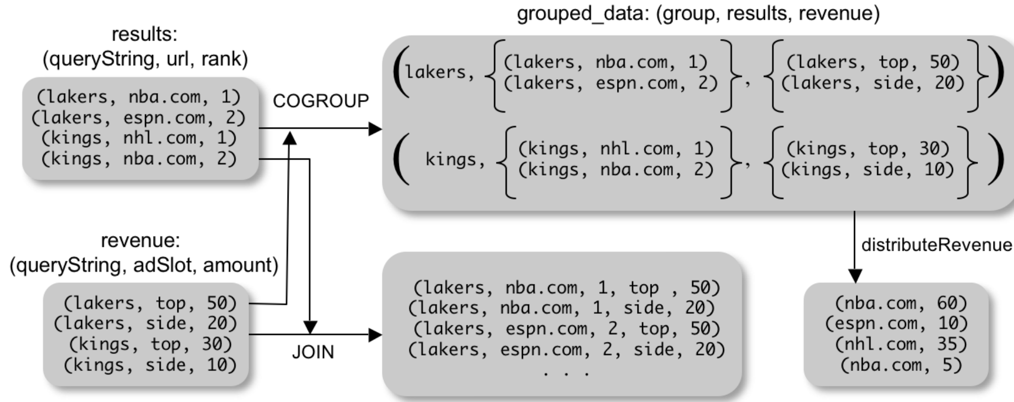

Figure 2: `COGROUP` versus `JOIN`.

string comparison, as opposed to numeric comparison which is specified via `==`. Filtering conditions in Pig Latin can involve a combination of expressions (Table 1), comparison operators such as `==, eq, !=, neq`, and the logical connectors `AND, OR,` and `NOT`.

Since arbitrary expressions are allowed, it follows that we can use UDFs while filtering. Thus, in our less ideal world, where bots don't identify themselves, we can use a sophisticated UDF (`isBot`) to perform the filtering, e.g.,

```
real_queries = FILTER queries BY NOT isBot(userId);
```

3.5 Getting Related Data Together: `COGROUP`

Per-tuple processing only takes us so far. It is often necessary to *group* together tuples from one or more data sets, that are related in some way, so that they can subsequently be processed together. This grouping operation is done by the `COGROUP` command. For example, suppose we have two data sets that we have specified through a `LOAD` command:

```
results:  (queryString, url, position)
revenue:  (queryString, adSlot, amount)
```

`results` contains, for different query strings, the urls shown as search results, and the position at which they were shown. `revenue` contains, for different query strings, and different advertisement slots, the average amount of revenue made by the advertisements for that query string at that slot. Then to group together all search result data and revenue data for the same query string, we can write:

```
grouped_data  =  COGROUP results BY queryString,
                         revenue BY queryString;
```

Figure 2 shows what a tuple in `grouped_data` looks like. In general, the output of a `COGROUP` contains one tuple for each group. The first field of the tuple (named `group`) is the group identifier (in this case, the value of the `queryString` field). Each of the next fields is a bag, one for each input being cogrouped, and is named the same as the alias of that input. The ith bag contains all tuples from the ith input belonging to that group. As in the case of filtering, grouping can also be performed according to arbitrary expressions which may include UDFs.

The reader may wonder why a `COGROUP` primitive is needed at all, since a very similar primitive is provided by the familiar, well-understood, `JOIN` operation in databases. For comparison, Figure 2 also shows the result of joining our data sets on `queryString`. It is evident that `JOIN` is equivalent to `COGROUP`, followed by taking a cross product of the tuples in the nested bags. While joins are widely applicable, certain custom processing might require access to the tuples of the groups before the cross-product is taken, as shown by the following example.

EXAMPLE 3. *Suppose we were trying to attribute search revenue to search-result urls to figure out the monetary worth of each url. We might have a sophisticated model for doing so. To accomplish this task in Pig Latin, we can follow the* `COGROUP` *with the following statement:*

```
url_revenues = FOREACH grouped_data GENERATE
      FLATTEN(distributeRevenue(results, revenue));
```

where `distributeRevenue` *is a UDF that accepts search results and revenue information for a query string at a time, and outputs a bag of urls and the revenue attributed to them. For example,* `distributeRevenue` *might attribute revenue from the top slot entirely to the first search result, while the revenue from the side slot may be attributed equally to all the results. In this case, the output of the above statement for our example data is shown in Figure 2.*

To specify the same operation in SQL, one would have to join by `queryString`*, then group by* `queryString`*, and then apply a custom aggregation function. But while doing the join, the system would compute the cross product of the search and revenue information, which the custom aggregation function would then have to undo. Thus, the whole process become quite inefficient, and the query becomes hard to read and understand.*

To reiterate, the `COGROUP` statement illustrates a key difference between Pig Latin and SQL. The `COGROUP` statements conforms to our goal of having an algebraic language, where each step carries out only a single transformation (Section 2.1). `COGROUP` carries out only the operation of grouping together tuples into nested bags. The user can subsequently choose to apply either an aggregation function

on those tuples, or cross-product them to get the join result, or process it in a custom way as in Example 3. In SQL, grouping is available only bundled with either aggregation (*group-by-aggregate* queries), or with cross-producting (the JOIN operation). Users find this additional flexibility of Pig Latin over SQL quite attractive, e.g.,

> "I frankly like pig much better than SQL in some respects (group + optional flatten works better for me, I love nested data structures)."
> – Ted Dunning, Chief Scientist, Veoh Networks

Note that it is our nested data model that allows us to have COGROUP as an independent operation—the input tuples are grouped together and put in nested bags. Such a primitive is not possible in SQL since the data model is flat. Of course, such a nested model raises serious concerns about efficiency of implementation: since groups can be very large (bigger than main memory, perhaps), we might build up gigantic tuples, which have these enormous nested bags within them. We address these efficiency concerns in our implementation section (Section 4).

3.5.1 *Special Case of* COGROUP: GROUP

A common special case of COGROUP is when there is only one data set involved. In this case, we can use the alternative, more intuitive keyword GROUP. Continuing with our example, if we wanted to find the total revenue for each query string, (a typical group-by-aggregate query), we can write it as follows:

```
grouped_revenue = GROUP revenue BY queryString;
query_revenues = FOREACH grouped_revenue GENERATE
                 queryString,
                 SUM(revenue.amount) AS totalRevenue;
```

In the second statement above, revenue.amount refers to a projection of the nested bag in the tuples of grouped_revenue. Also, as in SQL, the AS clause is used to assign names to fields on the fly.

To group *all* tuples of a data set together (e.g., to compute the overall total revenue), one uses the syntax GROUP revenue ALL.

3.5.2 JOIN *in Pig Latin*

Not all users need the flexibility offered by COGROUP. In many cases, all that is required is a regular equi-join. Thus, Pig Latin provides a JOIN keyword for equi-joins. For example,

```
join_result  =  JOIN results BY queryString,
                revenue BY queryString;
```

It is easy to verify that JOIN is only a syntactic shortcut for COGROUP followed by flattening. The above join command is equivalent to:

```
   temp_var  =  COGROUP results BY queryString,
                revenue BY queryString;
join_result  =  FOREACH temp_var GENERATE
                FLATTEN(results), FLATTEN(revenue);
```

3.5.3 *Map-Reduce in Pig Latin*

With the GROUP and FOREACH statements, it is trivial to express a map-reduce [4] program in Pig Latin. Converting to our data-model terminology, a map function operates on one input tuple at a time, and outputs a bag of key-value pairs. The reduce function then operates on all values for a key at a time to produce the final result. In Pig Latin,

```
map_result = FOREACH input GENERATE FLATTEN(map(*));
key_groups = GROUP map_result BY $0;
    output = FOREACH key_groups GENERATE reduce(*);
```

The first line applies the map UDF to every tuple on the input, and flattens the bag of key value pairs that it produces. (We use the shorthand * as in SQL to denote that all the fields of the input tuples are passed to the map UDF.) Assuming the first field of the map output to be the key, the second statement groups by key. The third statement then passes the bag of values for every key to the reduce UDF to obtain the final result.

3.6 Other Commands

Pig Latin has a number of other commands that are very similar to their SQL counterparts. These are:

1. UNION: Returns the union of two or more bags.
2. CROSS: Returns the cross product of two or more bags.
3. ORDER: Orders a bag by the specified field(s).
4. DISTINCT: Eliminates duplicate tuples in a bag. This command is just a shortcut for grouping the bag by all fields, and then projecting out the groups.

These commands are used as one would expect. For example, continuing with the example of Section 3.5.1, to order the query strings by their revenue:

```
ordered_result  =  ORDER query_revenues BY
                   totalRevenue;
```

3.7 Nested Operations

Each of the Pig Latin processing commands described so far operate over one or more bags of tuples as input. As illustrated in the previous sections, these commands collectively form a very powerful language. When we have nested bags within tuples, either as a result of (co)grouping, or due to the base data being nested, we might want to harness the same power of Pig Latin to process even these nested bags. To allow such processing, Pig Latin allows some commands to be nested within a FOREACH command.

For example, continuing with the data set of Section 3.5, suppose we wanted to compute for each queryString, the total revenue due to the 'top' ad slot, and also the overall total revenue. This can be written in Pig Latin as follows:

```
grouped_revenue = GROUP revenue BY queryString;
query_revenues = FOREACH grouped_revenue{
                    top_slot = FILTER revenue BY
                               adSlot eq 'top';
                    GENERATE queryString,
                             SUM(top_slot.amount),
                             SUM(revenue.amount);
                 };
```

In the above Pig Latin fragment, `revenue` is first grouped by `queryString` as before. Then each group is processed by a FOREACH command, within which is a `FILTER` command that operates on the nested bags on the tuples of `grouped_revenue`. Finally the `GENERATE` statement within the `FOREACH` outputs the required fields.

At present, we only allow `FILTER`, `ORDER`, and `DISTINCT` to be nested within `FOREACH`. In the future, as need arises, we might allow other constructs to be nested as well.

3.8 Asking for Output: STORE

The user can ask for the result of a Pig Latin expression sequence to be materialized to a file, by issuing the `STORE` command, e.g.,

```
STORE query_revenues INTO 'myoutput'
      USING myStore();
```

The above command specifies that bag `query_revenues` should be serialized to the file `myoutput` using the custom serializer `myStore`. As with `LOAD`, the `USING` clause may be omitted for a default serializer that writes plain text, tab-delimited files. Our system also comes with a built-in serializer/deserializer that can load/store arbitrarily nested data.

4. IMPLEMENTATION

Pig Latin is fully implemented by our system, *Pig*. Pig is architected to allow different systems to be plugged in as the execution platform for Pig Latin. Our current implementation uses *Hadoop* [10], an open-source, scalable implementation of map-reduce [4], as the execution platform. Pig Latin programs are compiled into map-reduce jobs, and executed using Hadoop. Pig, along with its Hadoop compiler, is an open-source project in the Apache incubator, and hence available for general use.

We first describe how Pig builds a logical plan for a Pig Latin program. We then describe our current compiler, that compiles a logical plan into map-reduce jobs executed using Hadoop. Last, we describe how our implementation avoids having large nested bags, and how it handles them if they do arise.

4.1 Building a Logical Plan

As clients issue Pig Latin commands, the Pig interpreter first parses it, and verifies that the input files and bags being referred to by the command are valid. For example, if the user enters `c = COGROUP a BY ..., b BY ...`, Pig verifies that the bags `a` and `b` have already been defined. Pig builds a *logical plan* for every bag that the user defines. When a new bag is defined by a command, the logical plan for the new bag is constructed by combining the logical plans for the input bags, and the current command. Thus, in the above example, the logical plan for `c` consists of a cogroup command having the logical plans for `a` and `b` as inputs.

Note that no processing is carried out when the logical plans are constructed. Processing is triggered only when the user invokes a `STORE` command on a bag. At that point, the logical plan for that bag is compiled into a physical plan, and is executed. This *lazy* style of execution is beneficial because it

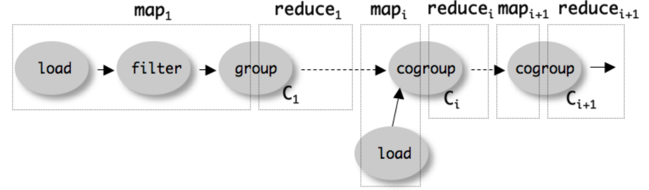

Figure 3: Map-reduce compilation of Pig Latin.

permits in-memory pipelining, and other optimizations such as filter reordering across multiple Pig Latin commands.

Pig is architected such that the parsing of Pig Latin and the logical plan construction is independent of the execution platform. Only the compilation of the logical plan into a physical plan depends on the specific execution platform chosen. Next, we describe the compilation into Hadoop map-reduce, the execution platform currently used by Pig.

4.2 Map-Reduce Plan Compilation

Compilation of a Pig Latin logical plan into map-reduce jobs is fairly simple. The map-reduce primitive essentially provides the ability to do a large-scale group by, where the map tasks assign keys for grouping, and the reduce tasks process a group at a time. Our compiler begins by converting each `(CO)GROUP` command in the logical plan into a distinct map-reduce job with its own map and reduce functions.

The map function for `(CO)GROUP` command C initially just assigns keys to tuples based on the `BY` clause(s) of C; the reduce function is initially a no-op. The map-reduce boundary is the cogroup command. The sequence of `FILTER`, and `FOREACH` commands from the `LOAD` to the first `COGROUP` operation C_1, are pushed into the map function corresponding to C_1 (see Figure 3). The commands that intervene between subsequent `COGROUP` commands C_i and C_{i+1} can be pushed into either (a) the reduce function corresponding to C_i, or (b) the map function corresponding to C_{i+1}. Pig currently always follows option (a). Since grouping is often followed by aggregation, this approach reduces the amount of data that has to be materialized between map-reduce jobs.

In the case of a `COGROUP` command with more than one input data set, the map function appends an extra field to each tuple that identifies the data set from which the tuple originated. The accompanying reduce function decodes this information and uses it to insert the tuple into the appropriate nested bag when cogrouped tuples are formed (recall Figure 2).

Parallelism for `LOAD` is obtained since Pig operates over files residing in the Hadoop distributed file system. We also automatically get parallelism for `FILTER` and `FOREACH` operations since for a given map-reduce job, several map and reduce instances are run in parallel. Parallelism for `(CO)GROUP` is achieved since the output from the multiple map instances is repartitioned in parallel to the multiple reduce instances.

The `ORDER` command is implemented by compiling into two map-reduce jobs. The first job samples the input to determine quantiles of the sort key. The second job range-

partitions the input according to the quantiles (thereby ensuring roughly equal-sized partitions), followed by local sorting in the reduce phase, resulting in a globally sorted file.

The inflexibility of the map-reduce primitive results in some overheads while compiling Pig Latin into map-reduce jobs. For example, data must be materialized and replicated on the distributed file system between successive map-reduce jobs. When dealing with multiple data sets, an additional field must be inserted in every tuple to indicate which data set it came from. However, the Hadoop map-reduce implementation does provide many desired properties such as parallelism, load-balancing, and fault-tolerance. Given the productivity gains to be had through Pig Latin, the associated overhead is often acceptable. Besides, there is the possibility of plugging in a different execution platform that can implement Pig Latin operations without such overheads.

4.3 Efficiency With Nested Bags

Recall Section 3.5. Conceptually speaking, our `(CO)GROUP` command places tuples belonging to the same group into one or more nested bags. In many cases, the system can avoid actually materializing these bags, which is especially important when the bags are larger than one machine's main memory.

One common case is where the user applies a distributive or algebraic [8] aggregation function over the result of a `(CO)GROUP` operation. (Distributive is a special case of algebraic, so we will only discuss algebraic functions.) An algebraic function is one that can be structured as a tree of subfunctions, with each leaf subfunction operating over a subset of the input data. If nodes in this tree achieve data reduction, then the system can keep the amount of data materialized in any single location small. Examples of algebraic functions abound: `COUNT`, `SUM`, `MIN`, `MAX`, `AVERAGE`, `VARIANCE`, although some useful functions are not algebraic, e.g., `MEDIAN`.

When Pig compiles programs into Hadoop map-reduce jobs, it uses Hadoop's *combiner* feature to achieve a two-tier tree evaluation of algebraic functions. Pig provides a special API for algebraic user-defined functions, so that custom user functions can take advantage of this important optimization.

Nevertheless, there still remain cases where `(CO)GROUP` is followed by something other than an algebraic UDF, e.g., the program in Example 3.5, where `distributeRevenue` is not algebraic. To cope with these cases, our implementation allows for nested bags to spill to disk. Our disk-resident bag implementation comes with database-style external sort algorithms to do operations such as sorting and duplicate elimination of the nested bags (recall Section 3.7).

5. DEBUGGING ENVIRONMENT

The process of constructing a Pig Latin program is typically an iterative one: The user makes an initial stab at writing a program, submits it to the system for execution, and inspects the output to determine whether the program had the intended effect. If not, the user revises the program and repeats this process. If programs take a long time to execute (e.g., because the data is large), this process can be very slow and inefficient.

To avoid this slowness, users often create a side data set consisting of a small sample of the original one, for experimentation. Unfortunately this method does not always work well. As a simple example, suppose the program performs an equijoin of tables `A(x,y)` and `B(x,z)` on attribute `x`. If the original data contains many distinct values for `x`, then it is unlikely that a small sample of `A` and a small sample of `B` will contain any matching `x` values [3]. Hence the join over the sample data set may well produce an empty result, even if the program is correct. Similarly, a program with a selective filter executed on a sample data set may produce an empty result. In general it can be difficult to test the semantics of a program over a sample data set.

Pig comes with a debugging environment called *Pig Pen*, which creates a side data set automatically, and in a manner that avoids the problems outlined in the previous paragraph. To avoid these problems successfully, the side data set must be tailored to the particular user program at hand. We refer to this dynamically-constructed side data set as a *sandbox data set*; we briefly describe how it is created in Section 5.1.

Pig Pen's user interface consists of a two-panel window as shown in Figure 4. The left-hand panel is where the user enters her Pig Latin commands. The right-hand panel is populated automatically, and shows the effect of the user's program on the sandbox data set. In particular, the intermediate bag produced by each Pig Latin command is displayed.

Suppose we have two data sets: a log of page visits, `visits: (user, url, time)`, and a catalog of pages and their pageranks, `pages: (url, pagerank)`. The program shown in Figure 4 finds web surfers who tend to visit high-pagerank pages. The program joins the two data sets after first running the log entries through a UDF that converts urls to a canonical form. After the join, the program groups tuples by user, computes the average pagerank for each user, and then filters users by average pagerank.

The right-hand panel of Figure 4 shows a sandbox data set, and how it is transformed by each successive command. The main semantics of each command are illustrated via the sandbox data set: We see that the `JOIN` command matches `visits` tuples with `pages` tuples on `url`. We also see that grouping by `user` creates one tuple per group, possibly containing multiple nested tuples as in the case of Amy. Lastly we see that the `FOREACH` command eliminates the nesting via aggregation, and that the `FILTER` command eliminates Fred, whose average pagerank is too low.

If one or more commands had been written incorrectly, e.g., if the user had forgotten to include `group` following `FOREACH`, the problem would be apparent in the right-hand panel. Similarly, if the program contains UDFs (as is common among real Pig users), the right-hand panel indicates whether the correct UDF is being applied, and whether it is behaving as intended. The sandbox data set also helps users understand the schema at each step, which is especially helpful in the presence of nested data.

In addition to helping the user spot bugs, this kind of interface facilitates writing a program in an incremental fashion: We write the first three commands and inspect the right-

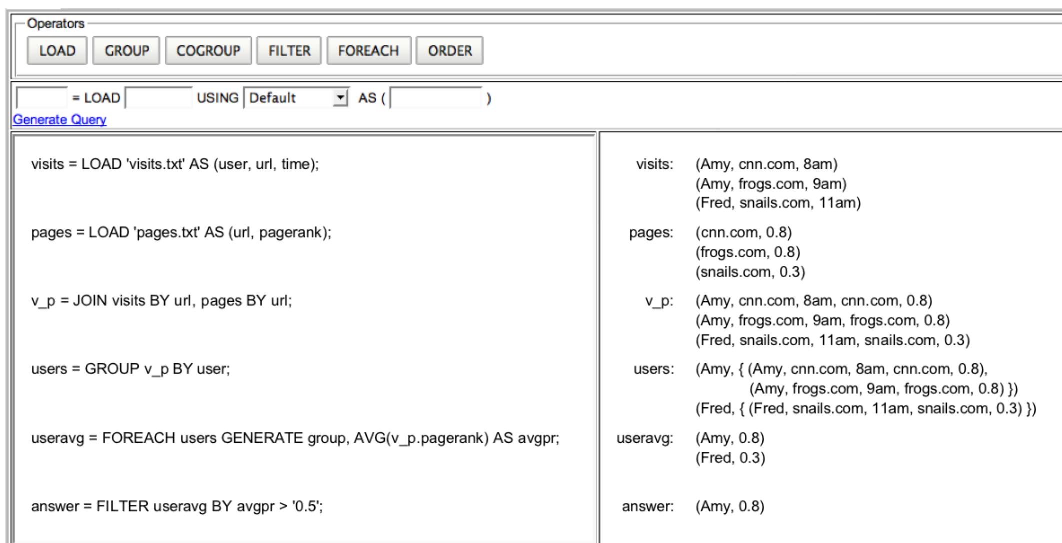

Figure 4: Pig Pen screenshot; displayed program finds users who tend to visit high-pagerank pages.

hand panel to ensure that we are joining the tables properly. Then we add the `GROUP` command and use the right-hand panel to help understand the schema of its output tuples. Then we proceed to add the `FOREACH` and `FILTER` commands, one at a time, until we arrive at the desired result. Once we are convinced the program is correct, we submit it for execution over the real data.

5.1 Generating a Sandbox Data Set

Pig Pen's sandbox data set generator takes as input a Pig Latin program P consisting of a sequence of n commands, where each command consumes one or more input bags and produces an output bag. The output of the data set generator is a set of *example bags* $\{B_1, B_2, \ldots, B_n\}$, one corresponding to the output of each command in P, as illustrated in Figure 4. The set of example bags is required to be *consistent*, meaning that the example output bag of each operator is exactly the bag produced by executing the command over its example input bag(s). The example bags that correspond to the `LOAD` commands comprise the *sandbox data set*.

There are three primary objectives in selecting a sandbox data set:

- **Realism.** The sandbox data set should be a subset of the actual data set, if possible. If not, then to the extent possible the individual data values should be ones found in the actual data set.
- **Conciseness.** The example bags should be as small as possible.
- **Completeness.** The example bags should collectively illustrate the key semantics of each command.

As an example of what is meant by the completeness objective, the example bags before and after the `GROUP` command in Figure 4 serve to illustrate the semantics of grouping by user, namely that input tuples about the same user are placed into a single output tuple. As another example, the example bags before and after the `FILTER` command illustrate the semantics of filtering by average pagerank, namely that input tuples with low average pagerank are not propagated to the output.

The procedure used in Pig Pen to generate a sandbox database starts by taking small random samples of the base data. Then, Pig Pen synthesizes additional data tuples to improve completeness. (When possible, the synthetic tuples are populated with real data values from the underlying domain, to minimize the impact on realism.) A final pruning pass eliminates redundant example tuples to improve conciseness. The details of the algorithm are beyond the scope of this paper.

6. USAGE SCENARIOS

In this section, we describe a sample of data analysis tasks that are being carried out at Yahoo! with Pig. Due to confidentiality reasons, we describe these tasks only at a high-level, and are not able to provide real Pig Latin programs. The use of Pig Latin ranges from group-by-aggregate and rollup queries (which are easy to write in SQL, and simple ones are also easy to write in map-reduce) to more complex tasks that use the full flexibility and power of Pig Latin.

Rollup aggregates: A common category of processing done using Pig involves computing various kinds of rollup aggregates against user activity logs, web crawls, and other data sets. One example is to calculate the frequency of search terms aggregated over days, weeks, or months, and also over geographical location as implied by the IP address. Some tasks require two or more successive aggregation passes, e.g., count the number of searches per user, and then compute the average per-user count. Other tasks require a join followed by aggregation, e.g., match search phrases with n-grams found in web page anchortext strings, and then count the number of matches per page. In such cases, Pig orchestrates a sequence of multiple map-reduce jobs on the user's behalf.

The primary reason for using Pig rather than a

database/OLAP system for these rollup analyses, is that the search logs are too big and continuous to be curated and loaded into databases, and hence are just present as (distributed) files. Pig provides an easy way to compute such aggregates directly over these files, and also makes it easy to incorporate custom processing, such as IP-to-geo mapping and n-gram extraction.

Temporal analysis: Temporal analysis of search logs mainly involves studying how search query distributions change over time. This task usually involves cogrouping search queries of one period with those of another period in the past, followed by custom processing of the queries in each group. The `COGROUP` command is a big win here since it provides access to the set of search queries in each group (recall Figure 2), making it easy and intuitive to write UDFs for custom processing. If a regular join were carried out instead, a cross product of search queries in the same group would be returned, leading to problems as in Example 3.

Session analysis: In session analysis, web user sessions, i.e., sequences of page views and clicks made by users, are analyzed to calculate various metrics such as: how long is the average user session, how many links does a user click on before leaving a website, how do click patterns vary in the course of a day/week/month. This analysis task mainly consists of grouping the activity logs by user and/or website, ordering the activity in each group by timestamp, and then applying custom processing on this sequence. In this scenario, our nested data model provides a natural abstraction for representing and manipulating sessions, and nested declarative operations such as order-by (Section 3.7) help minimize custom code and avoid the need for out-of-core algorithms inside UDFs in case of large groups.

7. RELATED WORK

We have discussed the relationship of Pig Latin to SQL and map-reduce throughout the paper. In this section, we compare Pig against other data processing languages and systems.

Pig is meant for offline, ad-hoc, scan-centric workloads. There is a family of recent, large-scale, distributed systems that provide database-like capabilites, but are geared toward transactional workloads and/or point lookups. Amazon's Dynamo [5], Google's BigTable [2], and Yahoo!'s PNUTS [9] are examples. Unlike Pig, these systems are not meant for data analysis. BigTable does have hooks through which data residing in BigTable can be analyzed using a map-reduce job, but in the end, the analysis engine is still map-reduce.

A variant of map-reduce that deals well with joins has been proposed [14], but it still does not deal with multi-stage programs.

Dryad [12] is a distributed platform that is being developed at Microsoft to provide large-scale, parallel, fault-tolerant execution of processing tasks. Dryad is more flexible than map-reduce as it allows the execution of arbitrary computation that can be expressed as directed acyclic graphs. Map-reduce, on the other hand, can only execute a simple, two-step chain of a map followed by a reduce. As mentioned in Section 4, Pig Latin is independent of the choice of the execution platform. Hence in principle, Pig Latin can be compiled into Dryad jobs. As with map-reduce, the Dryad layer is hard to program to. Hence Dryad has its own high-level language called DryadLINQ [6]. Little is known publicly about the language except that it is "SQL-like."

Sawzall [13] is a scripting language used at Google on top of map-reduce. Like map-reduce, a Sawzall program also has a fairly rigid structure consisting of a filtering phase (the map step) followed by an aggregation phase (the reduce step). Furthermore, only the filtering phase can be written by the user, and only a pre-built set of aggregations are available (new ones are non-trivial to add). While Pig Latin has similar higher level primitives like filtering and aggregation, an arbitrary number of them can be flexibly chained together in a Pig Latin program, and all primitives can use user-defined functions with equal ease. Further, Pig Latin has additional primitives such as cogrouping, that allow operations such as joins (which require multiple programs in Sawzall) to be written in a single line in Pig Latin.

We are certainly not the first to consider nested data models. Nested data models have been explored before in the context of object-oriented databases [11]. The programming languages community has explored data-parallel languages over nested data, e.g., NESL [1], to which Pig Latin bears some resemblance. Just as with map-reduce and Sawzall, NESL lacks support for combining multiple data sets (e.g., cogroup and join).

As for our debugging environment, we are not aware of any prior work on automatically generating intermediate example data for the purpose of illustrating processing semantics.

8. FUTURE WORK

Pig is a project under active development. We are continually adding new features to improve the user experience and yield even higher productivity gains. There are a number of promising directions that are yet unexplored in the context of the Pig system.

"Safe" optimizer: One of the arguments that we have put forward in the favor of Pig Latin is that due to its procedural nature, users have tighter control over how their programs are executed. However, we do not want to ignore database-style optimization altogether since it can often lead to huge improvements in performance. What is needed is a "safe" optimizer that will only perform optimizations that almost surely yield a performance benefit. In other cases, when the performance benefit is uncertain, or depends on unknown data characteristics, it should simply follow the Pig Latin sequence written by the user.

User interfaces: The productivity obtained by Pig can be enhanced through the right user interfaces. Pig Pen is a first step in that direction. Another idea worth exploring is to have a "boxes-and-arrows" GUI for specifying and viewing Pig Latin programs, in addition to the current textual language. A boxes-and-arrows paradigm illustrates the data flow structure of a program very explicitly, as reveals dependencies among the various computation steps. The UI should also promote sharing and collaboration, e.g., one should easily be able to link to a fragment of another user's

program, and incorporate UDFs provided by other users.

External functions: Pig programs currently run as Java map-reduce jobs, and consequently only support UDFs written in Java. However, for quick, ad-hoc tasks, the average user wants to write UDFs in a scripting language such as Perl or Python, instead of in a full-blown programming language like Java. Support for such functions requires a light-weight serialization/deserialization layer with bindings in the languages that we wish to support. Pig can then serialize data using this layer and pass it to the external process that runs the non-Java UDF, where it can be deserialized into that language's data structures. We are in the process of building such a layer and integrating it with Pig.

Unified environment: Pig Latin does not have control structures like loops and conditionals. If those are needed, Pig Latin, just like SQL, can be embedded in Java with a JDBC-style interface. However, as with SQL, this embedding can be awkward and difficult to work with. For example, all Pig Latin programs must be enclosed in strings, thus preventing static syntax checking. Results need to be converted back and forth between Java's data model and that of Pig. Moreover, the user now has to deal with three distinct environments: (a) the program in which the Pig Latin commands are embedded, (b) the Pig environment that understands Pig Latin commands, and (c) the programming language environment used to write the UDFs. We are exploring the idea of having a single, unified environment that supports development at all three levels. For this purpose, rather than designing yet another language, we want to embed Pig Latin in established languages such as Perl and Python[3] by making the language parsers aware of Pig Latin and able to package executable units for remote execution.

9. SUMMARY

We described a new data processing environment being deployed at Yahoo! called Pig, and its associated language, Pig Latin. Pig's target demographic is experienced procedural programmers who prefer map-reduce style programming over the more declarative, SQL-style programming, for stylistic reasons as well as the ability to control the execution plan. Pig aims for a sweet spot between these two extremes, offering high-level data manipulation primitives such as projection and join, but in a much less declarative style than SQL.

We also described a novel debugging environment we are developing for Pig, called Pig Pen. In conjunction with the step-by-step nature of our Pig Latin language, Pig Pen makes it easy and fast for users to construct and debug their programs in an incremental fashion. The user can write a prefix of their overall program, examine the output on the sandbox data set, and iterate until the output matches what they intended. At this point the user can "freeze" the program prefix, and begin to append further commands, without worrying about regressing on the progress made so far.

While Pig Pen is still in early stages of development, the core Pig system is fully implemented and available as an open-source Apache incubator project. The Pig system compiles Pig Latin expressions into a sequence of map-reduce jobs, and orchestrates the execution of these jobs on Hadoop, an open-source scalable map-reduce implementation. Pig has an active and growing user base inside Yahoo!, and with our recent open-source release we are beginning to attract users in the broader community.

[3]Such variants would obviously be called Erlpay and Ythonpay.

Acknowledgments

We are grateful to the Hadoop and Pig engineering teams at Yahoo! for helping us make Pig real. In particular we would like to thank Alan Gates and Olga Natkovich, the Pig engineering leads, for their invaluable contributions.

10. REFERENCES

[1] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[2] F. Chang et al. Bigtable: A distributed storage system for structured data. In *Proc. OSDI*, pages 205–218. USENIX Association, 2006.

[3] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proc. ACM SIGMOD*, 1999.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[5] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, 2007.

[6] Dryad LINQ. http://research.microsoft.com/research/sv/DryadLINQ/, 2007.

[7] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 1989.

[8] J. Gray et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[9] C. S. Group. Community Systems Research at Yahoo! *SIGMOD Record*, 36(3):47–54, September 2007.

[10] Hadoop. http://lucene.apache.org/hadoop/, 2007.

[11] R. Hull. A survey of theoretical research on typed complex database objects. In *XP7.52 Workshop on Database Theory*, 1986.

[12] M. Isard et al. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, Lisbon, Portugal, March 21-23 2007.

[13] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal*, 13(4), 2005.

[14] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: Simplified relational data processing on large clusters. In *Proc. ACM SIGMOD*, 2007.